

Cours Programmation Réseau : Socket TCP/UDP

© 2012-2020 <tvaira@free.fr> - v.1.2

Sommaire

L'interface socket	3
Pré-requis	3
Définition	3
Manuel du programmeur	3
Modèle	4
Couche Transport	5
Numéro de ports	5
Caractéristiques des sockets	6
Programmation TCP (Linux)	6
Objectifs	6
Diagramme d'échanges	6
Étape n°1 : création de la socket (côté client)	6
Étape n°2 : connexion au serveur	8
Étape n°3 : vérification du bon fonctionnement de la connexion	11
Étape n°4 : échange des données	11
Étape n°5 : réalisation d'un serveur TCP	15
Étape n°6 : mise en attente des connexions	18
Étape n°7 : accepter les demandes connexions	20
Programmation UDP (Linux)	25
Objectifs	25
Diagramme d'échanges	25
Étape n°1 : création de la socket (côté client)	25
Étape n°2 : attachement local de la socket	27
Étape n°3 : communication avec le serveur	29
Étape n°4 : vérification du bon fonctionnement de l'échange	33
Étape n°5 : réalisation d'un serveur UDP	33

Programmation Socket TCP (Windows)	37
Objectifs	37
Étape n°0 : préparation	37
Étape n°1 : création de la socket (côté client)	38
Étape n°2 : connexion au serveur	39
Étape n°3 : vérification du bon fonctionnement de la connexion	42
Étape n°4 : échange des données	42
Étape n°5 : réalisation d'un serveur TCP	46
Étape n°6 : mise en attente des connexions	48
Étape n°7 : accepter les demandes connexions	50
 Programmation TCP (Linux) IPv6	 55
 Questions de révision	 60
 Annexe 1 : le résolveur nom → adresse	 61

L'interface socket

Pré-requis

La mise en oeuvre de l'interface socket nécessite de connaître :

- L'architecture client/serveur
- L'adressage IP et les numéros de port
- Notions d'API (appels systèmes sous Unix) et de programmation en langage C
- Les protocoles TCP et UDP, les modes connecté et non connecté

Définition

« La notion de socket a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (Berkeley Software Distribution).

Un socket représente une **interface de communication logicielle** avec le système d'exploitation qui permet d'exploiter les services d'un protocole réseau et par laquelle une application peut envoyer et recevoir des données.

C'est donc un **mécanisme de communication bidirectionnelle entre processus** (locaux et/ou distants).

Un socket désigne aussi un **ensemble normalisé de fonctions de communication** (une API) qui est proposé dans quasiment tous les langages de programmation populaires (C, Java, C#, C++, ...) et répandue dans la plupart des systèmes d'exploitation (UNIX/Linux, ©Windows, ...).

Manuel du programmeur

Le développeur utilisera donc concrètement une interface pour programmer une application TCP/IP grâce par exemple :

- à l'API **Socket BSD** sous Unix/Linux ou
- à l'API **WinSocket** sous Microsoft ©Windows

Les pages man principales sous Unix/Linux concernant la programmation réseau sont regroupées dans le chapitre 7 :

- `socket(7)` : interface de programmation des sockets
- `packet(7)` : interface par paquet au niveau périphérique
- `raw(7)` : sockets brutes (raw) IPv4 sous Linux
- `ip(7)` : implémentation Linux du protocole IPv4
- `udp(7)` : protocole UDP pour IPv4
- `tcp(7)` : protocole TCP



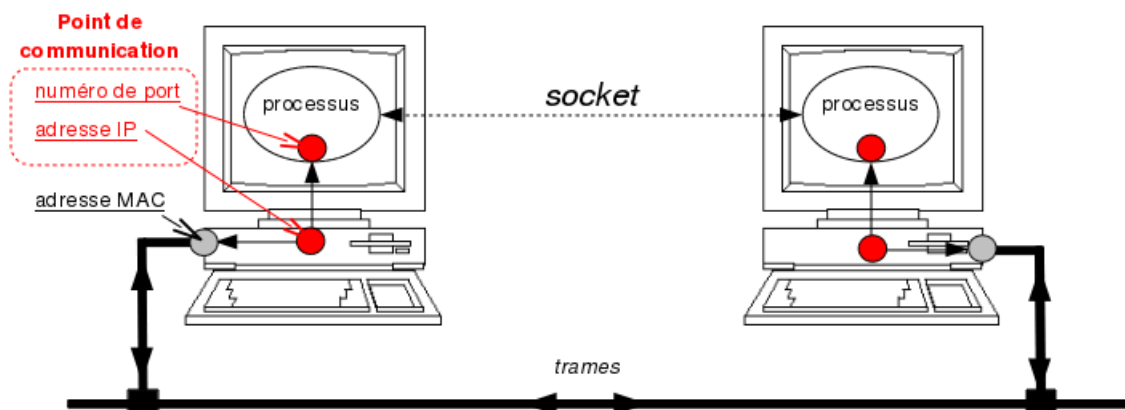
L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 7 socket`

Pour Microsoft ©Windows, on pourra utiliser le service en ligne MSDN :

- Windows Socket 2 : [msdn.microsoft.com/en-us/library/ms740673\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx)
- Les fonctions Socket : [msdn.microsoft.com/en-us/library/ms741394\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms741394(VS.85).aspx)

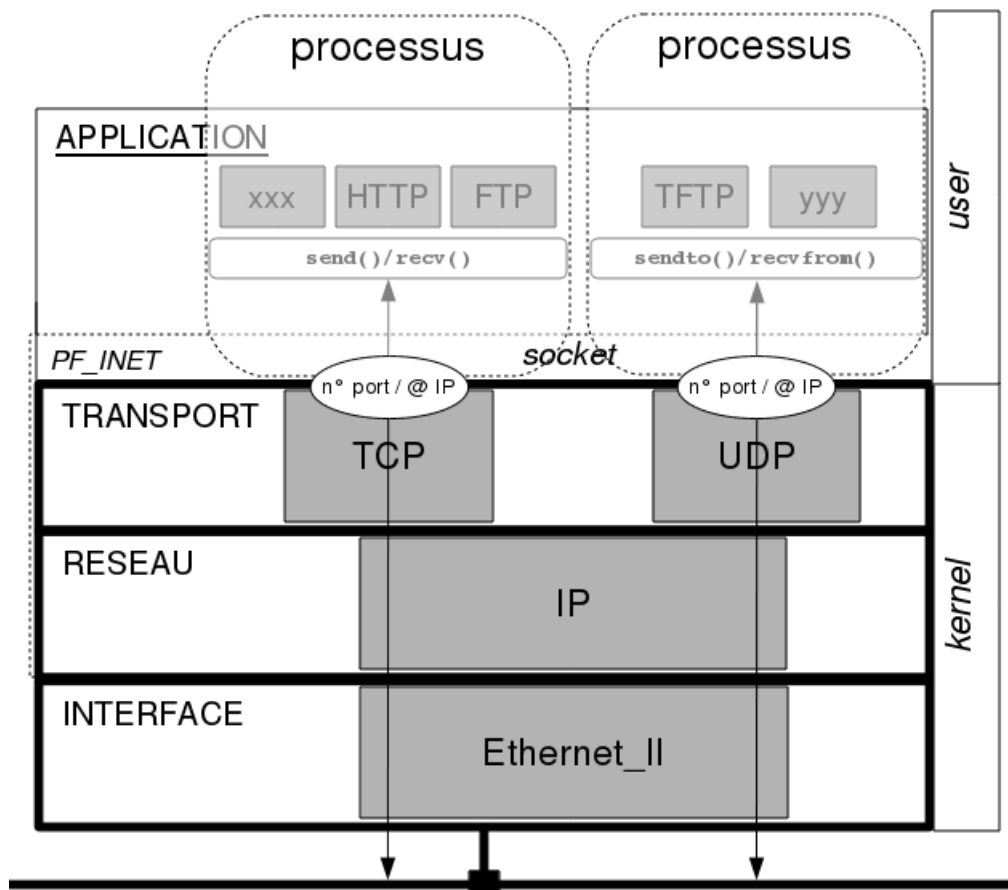
Modèle

Rappel : une socket est un point de communication par lequel un processus peut émettre et recevoir des données.



Ce point de communication devra être relié à une adresse IP et un numéro de port dans le cas des protocoles Internet.

Une socket est communément représentée comme un point d'entrée initial au niveau TRANSPORT du modèle à couches DoD dans la pile de protocole.



Exemple de processus TCP et UDP

Couche Transport

Rappel : la couche Transport est responsable du transport des messages complets de bout en bout (soit de processus à processus) au travers du réseau.

En programmation, si on utilise comme point d'entrée initial le niveau TRANSPORT, il faudra alors choisir un des deux protocoles de cette couche :

- **TCP** (*Transmission Control Protocol*) est un protocole de transport fiable, en **mode connecté** (RFC 793).
- **UDP** (*User Datagram Protocol*) est un protocole souvent décrit comme étant non-fiable, en **mode non-connecté** (RFC 768), mais plus rapide que TCP.

Numéro de ports

Rappel : un numéro de port sert à identifier un processus (l'application) en cours de communication par l'intermédiaire de son protocole de couche application (associé au service utilisé, exemple : 80 pour HTTP).



Pour chaque port, un numéro lui est attribué (codé sur 16 bits), ce qui implique qu'il existe un maximum de 65 536 ports (2^{16}) par machine et par protocoles TCP et UDP.

L'attribution des ports est faite par le système d'exploitation, sur demande d'une application. Ici, il faut distinguer les deux situations suivantes :

- cas d'un **processus client** : le numéro de port utilisé par le client sera envoyé au processus serveur. Dans ce cas, le processus client peut demander à ce que le système d'exploitation lui attribue n'importe quel port, à condition qu'il ne soit pas déjà attribué.
- cas d'un **processus serveur** : le numéro de port utilisé par le serveur doit être connu du processus client. Dans ce cas, le processus serveur doit demander un numéro de port précis au système d'exploitation qui vérifiera seulement si ce numéro n'est pas déjà attribué.



Une liste des ports dits réservés est disponible dans le fichier `/etc/services` sous Unix/Linux.

Caractéristiques des sockets

Rappel : les sockets compatibles BSD représentent une interface uniforme entre le processus utilisateur (user) et les piles de protocoles réseau dans le noyau (kernel) de l'OS.

Pour dialoguer, chaque processus devra préalablement créer une socket de communication en indiquant :

- le **domaine** de communication : ceci sélectionne la famille de protocole à employer. Il faut savoir que chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine `PF_INET` ou `AF_INET` et `AF_INET6` pour le protocole IPv6.
- le **type** de socket à utiliser pour le dialogue. Pour `PF_INET`, on aura le choix entre : `SOCK_STREAM` (qui correspond à un mode connecté donc TCP par défaut), `SOCK_DGRAM` (qui correspond à un mode non connecté donc UDP) ou `SOCK_RAW` (qui permet un accès direct aux protocoles de la couche Réseau comme IP, ICMP, ...).
- le **protocole** à utiliser sur la socket. Le numéro de protocole dépend du domaine de communication et du type de la socket. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée (`SOCK_STREAM` → TCP et `SOCK_DGRAM` → UDP). Néanmoins, rien ne s'oppose à ce que plusieurs protocoles existent, auquel cas il est nécessaire de le spécifier (c'est la cas pour `SOCK_RAW` où il faudra préciser le protocole à utiliser).



Une socket appartient à une famille. Il existe plusieurs types de sockets. Chaque famille possède son adressage.

Programmation TCP (Linux)

Objectifs

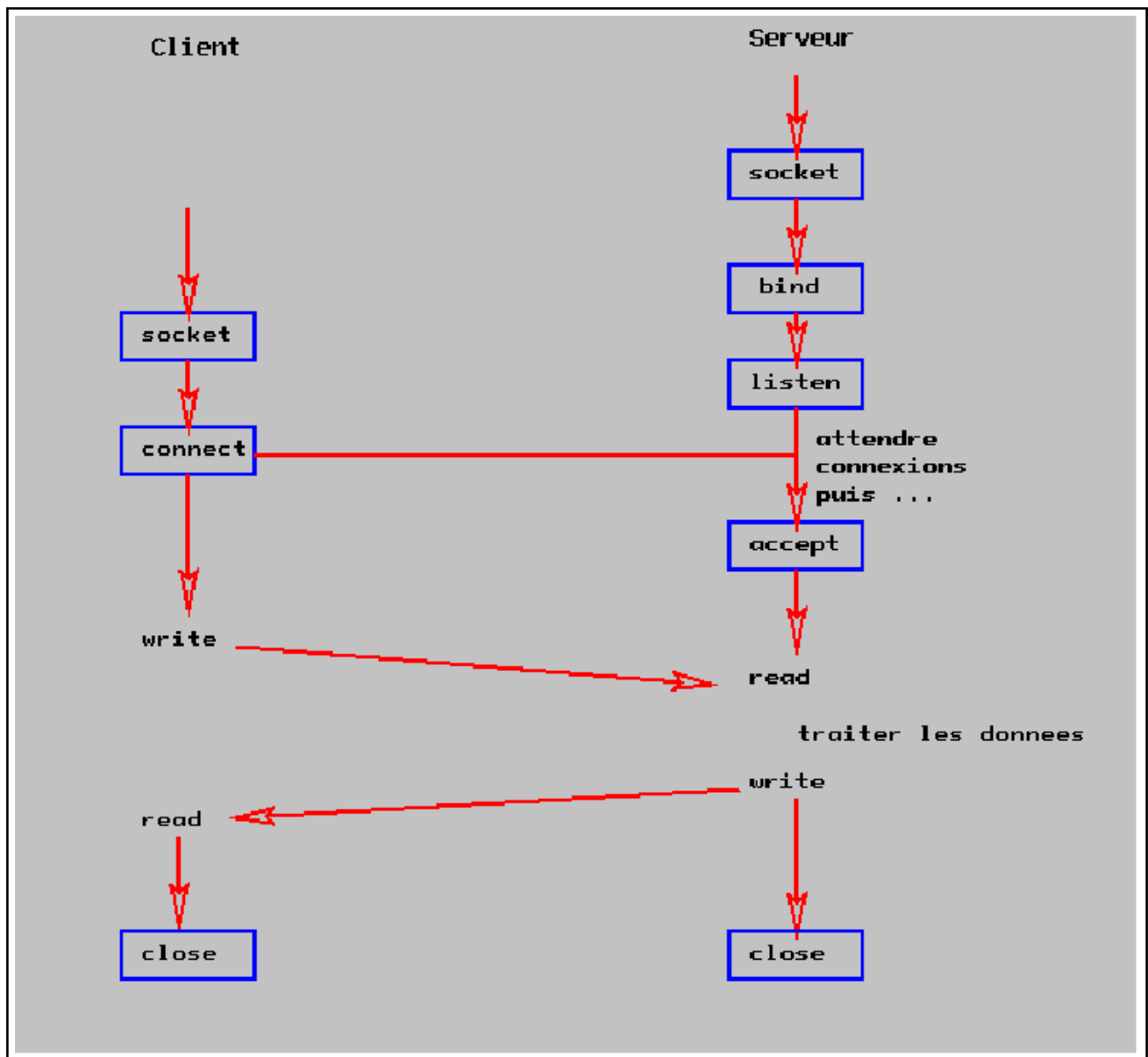
L'objectif de cette partie est la mise en oeuvre d'une communication client/serveur en utilisant une socket TCP sous Unix/Linux.

Diagramme d'échanges

L'échange entre un client et un serveur TCP peut être schématisé de la manière suivante :

Étape n°1 : création de la socket (côté client)

Pour créer une socket, on utilisera l'appel système `socket()`. On commence par consulter la page du manuel associée à cet appel :



Les appels systèmes utilisés dans un échange TCP

```
$ man 2 socket
```

```
SOCKET(2)
```

```
Manuel du programmeur Linux
```

```
SOCKET(2)
```

```
NOM
```

```
socket - Créer un point de communication
```

```
SYNOPSIS
```

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

```
DESCRIPTION
```

```
socket() crée un point de communication, et renvoie un descripteur.
```

```
...
```

```
VALEUR RENVOYÉE
```

Cet appel système renvoie un descripteur référençant la socket créée s'il réussit. S'il échoue, il renvoie -1 et errno contient le code d'erreur.

...

Extrait de la page man de l'appel système socket

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>

int main()
{
    int descripteurSocket;

    //--- Début de l'étape n°1 :
    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    //--Fin de l'étape n°1 !
    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    // On ferme la ressource avant de quitter
    close(descripteurSocket);

    return 0;
}
```

Étape n°1 : création de la socket (côté client)



Pour le paramètre `protocol`, on a utilisé la valeur 0 (voir commentaire). On aurait pu préciser le protocole TCP de la manière suivante : `IPPROTO_TCP`.

Étape n°2 : connexion au serveur

Maintenant que nous avons créé une socket TCP, il faut la connecter au processus serveur distant. Pour cela, on va utiliser l'appel système `connect()`. On consulte la page du manuel associée à cet appel :

```
$ man 2 connect
```

CONNECT(2)

Manuel du programmeur Linux

CONNECT(2)

NOM

`connect` - Débuter une connexion sur une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

DESCRIPTION

L'appel système `connect()` connecte la socket référencée par le descripteur de fichier `sockfd` à l'adresse indiquée par `serv_addr`. ...

VALEUR RENVOYÉE

`connect()` renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur. ...

Extrait de la page man de l'appel système connect

On rappelle que l'adressage du processus distant dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `PF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

L'interface socket propose une structure d'adresse générique :

```
struct sockaddr
{
    unsigned short int sa_family; //au choix
    unsigned char sa_data[14]; //en fonction de la famille
};
```

La structure générique sockaddr

Et le domaine `PF_INET` utilise une structure compatible :

// Remarque : ces structures sont déclarées dans `<netinet/in.h>`

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)

struct sockaddr_in
{
    unsigned short int sin_family; // <- PF_INET
    unsigned short int sin_port; // <- numéro de port
    struct in_addr sin_addr; // <- adresse IPv4
    unsigned char sin_zero[8]; // ajustement pour être compatible avec sockaddr
};
```

La structure compatible sockaddr_in pour PF_INET

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations distantes du serveur (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `inet_aton()` pour convertir une **adresse IP** depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau)

- `htons()` pour convertir le **numéro de port** (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



L'ordre des octets du réseau est en fait *big-endian*. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_STREAM, 0);

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    //--- Début de l'étape n°2 :
    // Obtient la longueur en octets de la structure sockaddr_in
    longueurAdresse = sizeof(pointDeRencontreDistant);
    // Initialise à 0 la structure sockaddr_in
    memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
    // Renseigne la structure sockaddr_in avec les informations du serveur distant
    pointDeRencontreDistant.sin_family = PF_INET;
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
    // On choisit l'adresse IPv4 du serveur
    inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier selon ses
        besoins

    // Débute la connexion vers le processus serveur distant
    if((connect(descripteurSocket, (struct sockaddr *)&pointDeRencontreDistant,
        longueurAdresse)) == -1)
    {
        perror("connect"); // Affiche le message d'erreur
        close(descripteurSocket); // On ferme la ressource avant de quitter
        exit(-2); // On sort en indiquant un code erreur
    }
}
```

```
//--- Fin de l'étape n°2 !
printf("Connexion au serveur réussie avec succès !\n");

// On ferme la ressource avant de quitter
close(descriptorSocket);
return 0;
}
```

Étape n°2 : connexion au serveur

Si vous testez ce client, vous risquez d'obtenir :

```
$ ./clientTCP-2
Socket créée avec succès ! (3)
connect: Connection refused
```

Ceci peut s'expliquer tout simplement parce qu'il n'y a pas de processus serveur à cette adresse !

Étape n°3 : vérification du bon fonctionnement de la connexion

Pour tester notre client, il nous faut un serveur ! Pour cela, on va utiliser l'outil réseau **netcat** en mode serveur (-l) sur le port 5000 (-p 5000) :

```
$ nc -l -p 5000
```

Puis :

```
$ ./clientTCP-2
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
```



Dans l'architecture client/serveur, on rappelle que c'est le client qui a l'initiative de l'échange. Il faut donc que le serveur soit en écoute avant que le client fasse sa demande.

Étape n°4 : échange des données

On rappelle qu'une communication TCP est bidirectionnelle *full duplex* et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.



Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide ! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

Les fonctions d'échanges de données sur une socket TCP sont :

- `read()` et `write()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket
- `recv()` et `send()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket avec un paramètre `flags`



Les appels `recv()` et `send()` sont spécifiques aux sockets en mode connecté. La seule différence avec `read()` et `write()` est la présence de `flags` (cf. man 2 `send`).

Faire communiquer deux processus sans aucun protocole de couche APPLICATION est tout de même difficile ! On va simplement fixer les règles d'échange suivantes :

- le client envoie en premier une chaîne de caractères
- et le serveur lui répondra "ok"

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

#define LG_MESSAGE 256

int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
    int ecrits, lus; /* nb d'octets ecrits et lus */
    int retour;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    // Obtient la longueur en octets de la structure sockaddr_in
    longueurAdresse = sizeof(pointDeRencontreDistant);
    // Initialise à 0 la structure sockaddr_in
    memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
    // Renseigne la structure sockaddr_in avec les informations du serveur distant
    pointDeRencontreDistant.sin_family = PF_INET;
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
    // On choisit l'adresse IPv4 du serveur
    inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier selon ses
        besoins

    // Débute la connexion vers le processus serveur distant
```

```

if((connect(descripteurSocket, (struct sockaddr *)&pointDeRencontreDistant,
    longueurAdresse)) == -1)
{
    perror("connect"); // Affiche le message d'erreur
    close(descripteurSocket); // On ferme la ressource avant de quitter
    exit(-2); // On sort en indiquant un code erreur
}

printf("Connexion au serveur réussie avec succès !\n");

//--- Début de l'étape n°4 :
// Initialise à 0 les messages
memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));

// Envoie un message au serveur
sprintf(messageEnvoi, "Hello world !\n");
ecrits = write(descripteurSocket, messageEnvoi, strlen(messageEnvoi)); // message à
    TAILLE variable
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("write");
        close(descripteurSocket);
        exit(-3);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le serveur !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* envoi de n octets */
        printf("Message %s envoyé avec succès (%d octets)\n\n", messageEnvoi, ecrits);
}

/* Reception des données du serveur */
lus = read(descripteurSocket, messageRecu, LG_MESSAGE*sizeof(char)); /* attend un message
    de TAILLE fixe */
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("read");
        close(descripteurSocket);
        exit(-4);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le serveur !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* réception de n octets */
        printf("Message reçu du serveur : %s (%d octets)\n\n", messageRecu, lus);
}

//--- Fin de l'étape n°4 !

// On ferme la ressource avant de quitter
close(descripteurSocket);

```

```
    return 0;
}
```

Étape n°4 : échange des données

On utilise la même procédure de test que précédemment en démarrant un serveur **netcat** sur le port 5000 :

```
$ nc -l -p 5000
```

Puis, on exécute notre client :

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
    envoyé avec succès (14 octets)

Message reçu du serveur : ok
    (3 octets)
```

Dans la console où on a exécuté le serveur **netcat**, cela donne :

```
$ nc -l -p 5000
Hello world !
ok
```



Dans **netcat**, pour envoyer des données au client, il suffit de saisir son message et de valider par la touche Entrée.

Que se passe-t-il si le serveur s'arrête (en tapant **Ctrl-C** par exemple!) au lieu d'envoyer "ok" ?

```
$ nc -l -p 5000
Hello world !
^C

$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
    envoyé avec succès (14 octets)
```

La socket a été fermée par le serveur !

Notre client a bien détecté la fermeture de la socket côté serveur.

Dans les codes sources ci-dessus, nous avons utilisés l'appel **close()** pour fermer la socket et donc la communication. En TCP, la communication étant bidirectionnelle *full duplex*, il est possible de fermer plus finement l'échange en utilisant l'appel **shutdown()** :

```
$ man 2 shutdown
```

SHUTDOWN(2)

Manuel du programmeur Linux

SHUTDOWN(2)

NOM

shutdown - Terminer une communication en full-duplex

SYNOPSIS

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

DESCRIPTION

La fonction shutdown() termine tout ou partie d'une connexion full-duplex sur la socket s. Si how vaut SHUT_RD, la réception est désactivée. Si how vaut SHUT_WR, l'émission est désactivée. Si how vaut SHUT_RDWR, l'émission et la réception sont désactivées.

VALEUR RENVOYÉE

Cet appel système renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

...

Extrait de la page man de l'appel système shutdown

Étape n°5 : réalisation d'un serveur TCP

Évidemment, un serveur TCP a lui aussi besoin de créer une socket SOCK_STREAM dans le domaine PF_INET.

Mis à part cela, le code source d'un serveur TCP basique est très différent d'un client TCP dans le principe. On va détailler ces différences étape par étape.

On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de processus client. Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.

Pour mettre en oeuvre cela, le serveur va utiliser l'appel système bind() qui va lui permettre de lier sa socket d'écoute à une interface et à un numéro de port local à sa machine.

\$ man 2 bind

BIND(2)

Manuel du programmeur Linux

BIND(2)

NOM

bind - Fournir un nom à une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

DESCRIPTION

Quand une socket est créée avec l'appel système socket(2), elle existe dans l'espace des noms mais n'a pas de nom assigné). bind() affecte l'adresse spécifiée dans addr à la socket référencée par le descripteur de fichier sockfd. addrlen indique la taille, en octets, de la structure d'adresse pointée par addr. Traditionnellement cette

opération est appelée « affectation d'un nom à une socket ».

Il est normalement nécessaire d'affecter une adresse locale avec `bind()` avant qu'une socket `SOCK_STREAM` puisse recevoir des connexions (voir `accept(2)`).

Les règles d'affectation de nom varient suivant le domaine de communication.

...

VALEUR RENVOYÉE

L'appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

...

Extrait de la page man de l'appel système `bind`

On rappelle que l'adressage d'un processus (local ou distant) dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `PF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

Rappel : l'interface socket propose une structure d'adresse générique `sockaddr` et le domaine `PF_INET` utilise une structure compatible `sockaddr_in`.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



Normalement il faudrait indiquer l'adresse IPv4 de l'interface locale du serveur qui acceptera les demandes de connexions. Il est ici possible de préciser avec `INADDR_ANY` que toutes les interfaces locales du serveur accepteront les demandes de connexion des clients.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */

#define PORT IPPORT_USERRESERVED // = 5000

int main()
{
    int socketEcoute;
```



```

struct sockaddr_in pointDeRencontreLocal;
socklen_t longueurAdresse;

//--- Début de l'étape n°5 :
// Crée un socket de communication
socketEcoute = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
    protocole par défaut associé à SOCK_STREAM soit TCP */

// Teste la valeur renvoyée par l'appel système socket()
if(socketEcoute < 0) /* échec ? */
{
    perror("socket"); // Affiche le message d'erreur
    exit(-1); // On sort en indiquant un code erreur
}

printf("Socket créée avec succès ! (%d)\n", socketEcoute);

// On prépare l'adresse d'attachement locale
longueurAdresse = sizeof(struct sockaddr_in);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
    locales disponibles
pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

// On demande l'attachement local de la socket
if((bind(socketEcoute, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
{
    perror("bind");
    exit(-2);
}

//--- Fin de l'étape n°5 !
printf("Socket attachée avec succès !\n");

// On s'endort ... (cf. test)
sleep(2);

// On ferme la ressource avant de quitter
close(socketEcoute);

return 0;
}

```

Étape n°5 : réalisation d'un serveur TCP

Le test est concluant :

```

$ ./serveurTCP-1
Socket créée avec succès ! (3)
Socket attachée avec succès !

```

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant !
Exécutons deux fois le même serveur et on obtient alors :

```

$ ./serveurTCP-1 & ./serveurTCP-1

```

```
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket créée avec succès ! (3)
bind: Address already in use
```



Explication : l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur.

Étape n°6 : mise en attente des connexions

Maintenant que le serveur a créé et attaché une socket d'écoute, il doit la placer en attente passive, c'est-à-dire capable d'accepter les demandes de connexion des processus clients.

Pour cela, on va utiliser l'appel système `listen()` :

```
$ man 2 listen
```

LISTEN(2)

Manuel du programmeur Linux

LISTEN(2)

NOM

`listen` - Attendre des connexions sur une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

DESCRIPTION

`listen()` marque la socket référencée par `sockfd` comme une socket passive, c'est-à-dire comme une socket qui sera utilisée pour accepter les demandes de connexions entrantes en utilisant `accept()`.

L'argument `sockfd` est un descripteur de fichier qui fait référence à une socket de type `SOCK_STREAM`.

L'argument `backlog` définit une longueur maximale jusqu'à laquelle la file des connexions en attente pour `sockfd` peut croître. Si une nouvelle connexion arrive alors que la file est pleine, le client reçoit une erreur indiquant `ECONNREFUSED`, ou, si le protocole sous-jacent supporte les retransmissions, la requête peut être ignorée afin qu'un nouvel essai réussisse.

VALEUR RENVOYÉE

Cet appel système renvoie 0 si il réussit, ou -1 en cas d'échec, auquel cas `errno` est renseignée en conséquence.

...

Extrait de la page man de l'appel système `listen`



Si la file est pleine, le serveur sera dans une situation de DOS (*Deny Of Service*) car il ne peut plus traiter les nouvelles demandes de connexion.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */

#define PORT IPPORT_USERRESERVED // = 5000

int main()
{
    int socketEcoule;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;

    // Crée un socket de communication
    socketEcoule = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(socketEcoule < 0) {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }
    printf("Socket créée avec succès ! (%d)\n", socketEcoule);

    // On prépare l'adresse d'attachement locale
    longueurAdresse = sizeof(struct sockaddr_in);
    memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
    pointDeRencontreLocal.sin_family = PF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
        locales disponibles
    pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

    // On demande l'attachement local de la socket
    if((bind(socketEcoule, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
    {
        perror("bind");
        exit(-2);
    }
    printf("Socket attachée avec succès !\n");

    //--- Début de l'étape n°6 :
    // On fixe la taille de la file d'attente à 5 (pour les demandes de connexion non encore
        traitées)
    if(listen(socketEcoule, 5) < 0)
    {
        perror("listen");
        exit(-3);
    }
}
```

```
//--- Fin de l'étape n°6 !
printf("Socket placée en écoute passive ...\n");

// On ferme la ressource avant de quitter
close(socketEcoute);
return 0;
}
```

Étape n°6 : mise en attente des connexions

Étape n°7 : accepter les demandes connexions

Cette étape est cruciale pour le serveur. Il lui faut maintenant accepter les demandes de connexion en provenance des processus client.

Pour cela, il va utiliser l'appel système `accept()` :

\$ man 2 accept

ACCEPT(2) Manuel du programmeur Linux ACCEPT(2)

NOM

`accept` - Accepter une connexion sur une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>
```

```
int accept(int sockfd, struct sockaddr *adresse, socklen_t *longueur);
```

DESCRIPTION

L'appel système `accept()` est employé avec les sockets utilisant un protocole en mode connecté `SOCK_STREAM`. Il extrait la première connexion de la file des connexions en attente de la socket `sockfd` à l'écoute, crée une nouvelle socket connectée, et renvoie un nouveau descripteur de fichier qui fait référence à cette socket.

La nouvelle socket n'est pas en état d'écoute. La socket originale `sockfd` n'est pas modifiée par l'appel système.

...

VALEUR RENVOYÉE

S'il réussit, `accept()` renvoie un entier non négatif, constituant un descripteur pour la nouvelle socket. S'il échoue, l'appel renvoie `-1` et `errno` contient le code d'erreur.

...

Extrait de la page man de l'appel système `accept`



Explication : imaginons qu'un client se connecte à notre socket d'écoute. L'appel `accept()` va retourner une nouvelle socket connectée au client qui servira de socket de dialogue. La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.

Le principe est simple mais un problème apparaît pour le serveur : comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ? Il y a plusieurs solutions à ce problème notamment la programmation multi-tâche car ici le serveur a besoin de paralléliser plusieurs traitements.

On va pour l'instant ignorer ce problème et mettre en oeuvre un serveur basique : c'est-à-dire mono-client (ou plus exactement un client après l'autre) !

Concernant le dialogue, on utilisera au choix les mêmes fonctions (`read()/write()` ou `recv()/send()`) que le client.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */

#define PORT IPPORT_USERRESERVED // = 5000

#define LG_MESSAGE 256

int main()
{
    int socketEcoule;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;
    int socketDialogue;
    struct sockaddr_in pointDeRencontreDistant;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    char messageReçu[LG_MESSAGE]; /* le message de la couche Application ! */
    int écrits, lus; /* nb d'octets écrits et lus */
    int retour;

    // Crée un socket de communication
    socketEcoule = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(socketEcoule < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    printf("Socket créée avec succès ! (%d)\n", socketEcoule);

    // On prépare l'adresse d'attachement locale
    longueurAdresse = sizeof(struct sockaddr_in);
    memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
    pointDeRencontreLocal.sin_family = PF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
        locales disponibles
    pointDeRencontreLocal.sin_port = htons(PORT);
```

```
// On demande l'attachement local de la socket
if((bind(socketEcoule, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
{
    perror("bind");
    exit(-2);
}

printf("Socket attachée avec succès !\n");

// On fixe la taille de la file d'attente à 5 (pour les demandes de connexion non encore
traitées)
if(listen(socketEcoule, 5) < 0)
{
    perror("listen");
    exit(-3);
}

printf("Socket placée en écoute passive ...\n");

//--- Début de l'étape n°7 :
// boucle d'attente de connexion : en théorie, un serveur attend indéfiniment !
while(1)
{
    memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
    memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));
    printf("Attente d'une demande de connexion (quitter avec Ctrl-C)\n\n");
    // c'est un appel bloquant
    socketDialogue = accept(socketEcoule, (struct sockaddr *)&pointDeRencontreDistant, &
        longueurAdresse);

    if (socketDialogue < 0)
    {
        perror("accept");
        close(socketDialogue);
        close(socketEcoule);
        exit(-4);
    }

    // On réception les données du client (cf. protocole)
    lus = read(socketDialogue, messageRecu, LG_MESSAGE*sizeof(char)); // ici appel
    bloquant
    switch(lus)
    {
        case -1 : /* une erreur ! */
            perror("read");
            close(socketDialogue);
            exit(-5);
        case 0 : /* la socket est fermée */
            fprintf(stderr, "La socket a été fermée par le client !\n\n");
            close(socketDialogue);
            return 0;
        default: /* réception de n octets */
            printf("Message reçu : %s (%d octets)\n\n", messageRecu, lus);
    }
}
```

```
}

// On envoie des données vers le client (cf. protocole)
sprintf(messageEnvoi, "ok\n");
ecrits = write(socketDialogue, messageEnvoi, strlen(messageEnvoi));
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("write");
        close(socketDialogue);
        exit(-6);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le client !\n\n");
        close(socketDialogue);
        return 0;
    default: /* envoi de n octets */
        printf("Message %s envoyé (%d octets)\n\n", messageEnvoi, ecrits);
}

// On ferme la socket de dialogue et on se replace en attente ...
close(socketDialogue);
}

//--- Fin de l'étape n°7 !

// On ferme la ressource avant de quitter
close(socketEcoute);

return 0;
}
```

Étape n°7 : accepter les demandes connexions

Testons notre serveur avec notre client :

```
$ ./serveurTCP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket placée en écoute passive ...
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

```
Message reçu : Hello world !
(14 octets)
```

```
Message ok
envoyé (3 octets)
```

```
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

```
Message reçu : Hello world !
(14 octets)
```

```
Message ok
envoyé (3 octets)
```

```
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

^C

On va exécuter deux clients à la suite :

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
  envoyé avec succès (14 octets)

Message reçu du serveur : ok
  (3 octets)
```

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
  envoyé avec succès (14 octets)

Message reçu du serveur : ok
  (3 octets)
```



Il est évidemment possible de tester notre serveur avec des clients TCP existants comme telnet ou netcat.

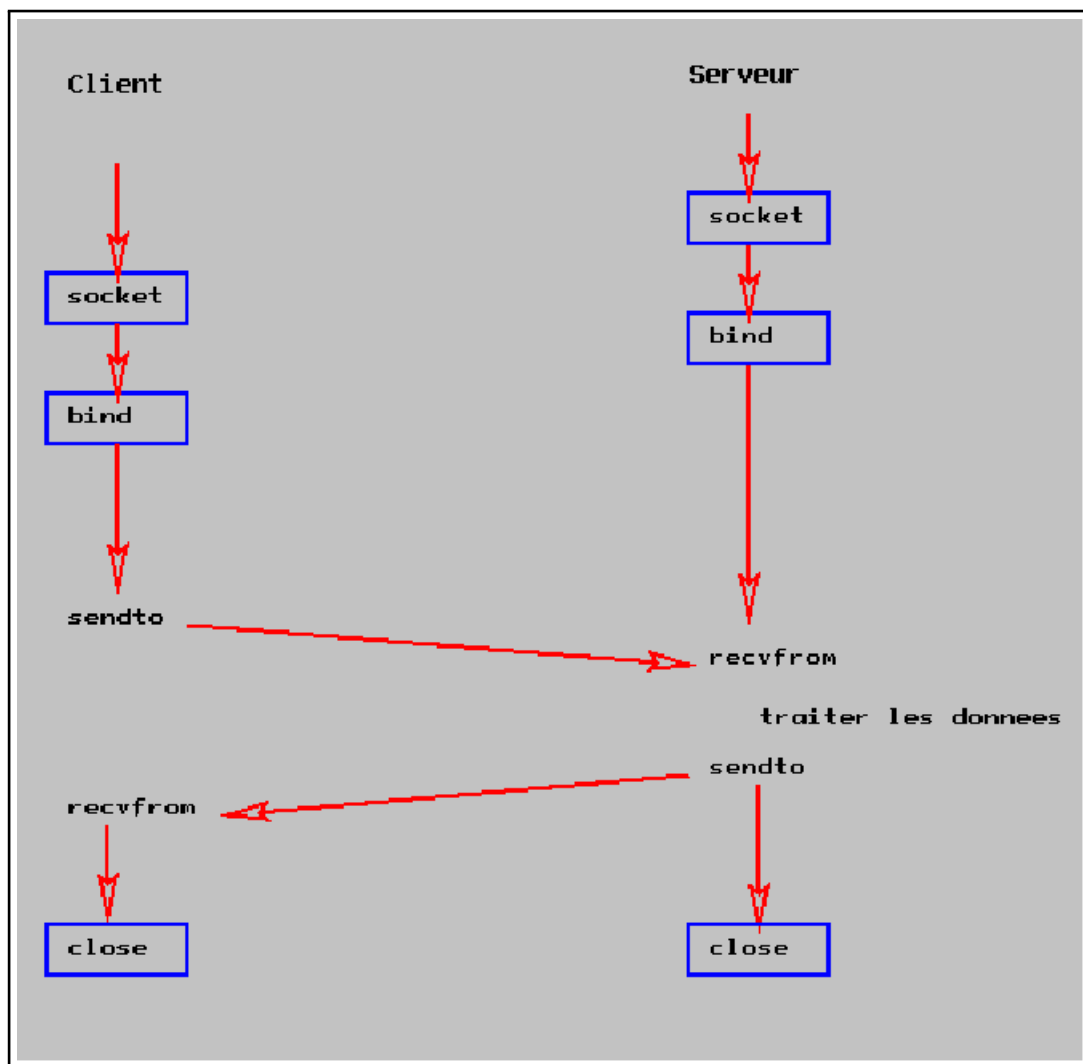
Programmation UDP (Linux)

Objectifs

L'objectif de cette partie est la mise en oeuvre d'une communication client/serveur en utilisant une **socket UDP** sous Unix/Linux.

Diagramme d'échanges

L'échange entre un client et un serveur UDP peut être schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange UDP

Étape n°1 : création de la socket (côté client)

Pour créer une socket, on utilisera l'appel système `socket()`. On commence par consulter la page du manuel associée à cet appel :

```
$ man 2 socket
```

NOM

socket - Créer un point de communication

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

DESCRIPTION

socket() crée un point de communication, et renvoie un descripteur.
...

VALEUR RENVOYÉE

Cet appel système renvoie un descripteur référençant la socket créée s'il réussit.
S'il échoue, il renvoie -1 et errno contient le code d'erreur.

...

Extrait de la page man de l'appel système socket

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>

int main()
{
    int descripteurSocket;

    //<-- Début de l'étape n°1 !
    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_DGRAM soit UDP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    //--> Fin de l'étape n°1 !
    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    // On ferme la ressource avant de quitter
    close(descripteurSocket);

    return 0;
}
```

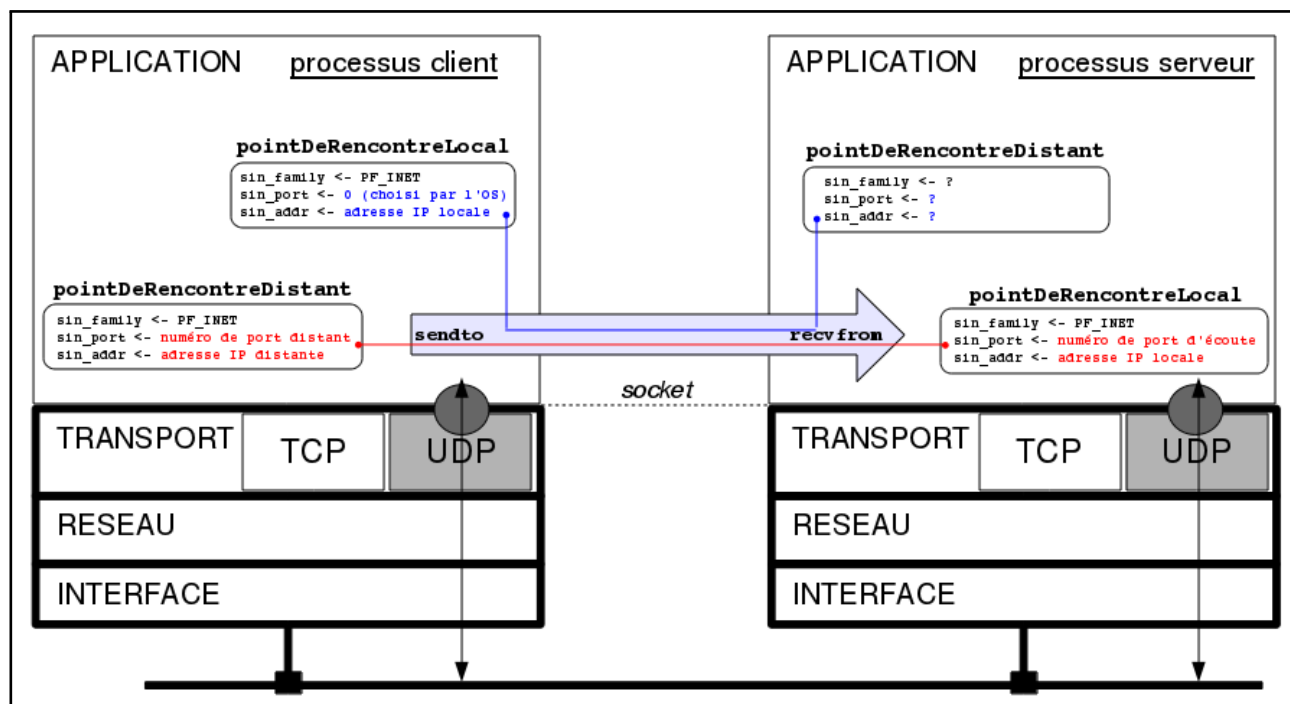
Étape n°1 : création de la socket (côté client)



Pour le paramètre `protocol`, on a utilisé la valeur 0 (voir commentaire). On aurait pu préciser le protocole UDP de la manière suivante : `IPPROTO_UDP`.

Étape n°2 : attachement local de la socket

Maintenant que nous avons créé une socket UDP, le client pourrait déjà communiquer avec un serveur UDP car nous utilisons un mode non-connecté.



Un échange en UDP

On va tout d'abord attacher cette socket à une interface et à un numéro de port local de sa machine en utilisant l'appel système `bind()`. Cela revient à créer un **point de rencontre local pour le client**. On consulte la page du manuel associée à cet appel :

```
$ man 2 bind
```

BIND(2)

Manuel du programmeur Linux

BIND(2)

NOM

`bind` - Fournir un nom à une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

DESCRIPTION

Quand une socket est créée avec l'appel système `socket(2)`, elle existe dans l'espace des noms mais n'a pas de nom assigné). `bind()` affecte l'adresse spécifiée dans `addr` à la socket référencée par le descripteur de fichier `sockfd`. `addrlen` indique la taille, en octets, de la structure d'adresse pointée par `addr`. Traditionnellement cette

opération est appelée « affectation d'un nom à une socket ».

Les règles d'affectation de nom varient suivant le domaine de communication.
...

VALEUR RENVOYÉE

L'appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

...

Extrait de la page man de l'appel système `bind`

On rappelle que l'adressage d'un processus (local ou distant) dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `PF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

Rappel : l'interface socket propose une structure d'adresse générique `sockaddr` et le domaine `PF_INET` utilise une structure compatible `sockaddr_in`.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du client** (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



Normalement, il faudrait indiquer un numéro de port utilisé par le client pour cette socket. Cela peut s'avérer délicat si on ne connaît pas les numéros de port libres. Le plus simple est de laisser le système d'exploitation choisir en indiquant la valeur 0 dans le champ `sin_port`.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et htonl */

int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0);
```

```
// Teste la valeur renvoyée par l'appel système socket()
if(descripteurSocket < 0) {
    perror("socket"); // Affiche le message d'erreur
    exit(-1); // On sort en indiquant un code erreur
}
printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

//<-- Début de l'étape n°2 !
// On prépare l'adresse d'attachement locale
longueurAdresse = sizeof(struct sockaddr_in);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // n'importe quelle interface
    locale disponible
pointDeRencontreLocal.sin_port = htons(0); // l'os choisira un numéro de port libre

// On demande l'attachement local de la socket
if((bind(descripteurSocket, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse))
    < 0) {
    perror("bind");
    exit(-2);
}
//--> Fin de l'étape n°2 !
printf("Socket attachée avec succès !\n");

// On ferme la ressource avant de quitter
close(descripteurSocket);
return 0;
}
```

Étape n°2 : attachement local de la socket

Le test est concluant :

```
$ ./clientUDP-2
Socket créée avec succès ! (3)
Socket attachée avec succès !
```

Étape n°3 : communication avec le serveur

Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.



Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide ! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

Les fonctions d'échanges de données sur une socket UDP sont `recvfrom()` et `sendto()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket en mode non-connecté.



Les appels `recvfrom()` et `sendto()` sont spécifiques aux sockets en mode non-connecté. Ils utiliseront en argument une structure `sockaddr_in` pour `PF_INET`.

Ici, on limitera notre client à l'envoi d'une chaîne de caractères. Pour cela, on va utiliser l'appel `sendto()` :

```
$ man 2 sendto
```

SEND(2)

Manuel du programmeur Linux

SEND(2)

NOM

`sendto` - Envoyer un message sur une socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
```

DESCRIPTION

L'appel système `sendto()` permet de transmettre un message à destination d'une autre socket.

Le paramètre `s` est le descripteur de fichier de la socket émettrice. L'adresse de la cible est fournie par `to`, `tolen` spécifiant sa taille. Le message se trouve dans `buf` et a pour longueur `len`.

...

VALEUR RENVOYÉE

S'ils réussissent, ces appels système renvoient le nombre de caractères émis. S'ils échouent, ils renvoient `-1` et `errno` contient le code d'erreur.

...

Extrait de la page man de l'appel système `sendto`

On rappelle que l'adressage du processus distant dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `PF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

Et il suffira donc d'initialiser une structure `sockaddr_in` avec les informations distantes du serveur (adresse IPv4 et numéro de port). Cela revient à adresser le **point de rencontre distant** qui sera utilisé dans l'appel `sendto()` par le client.

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `inet_aton()` pour convertir une adresse IP depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau)
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



L'ordre des octets du réseau est en fait *big-endian*. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

```

#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons, htonl et inet_aton */

#define LG_MESSAGE 256

int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreLocal;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    int ecrits; /* nb d'octets ecrits */
    int retour;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_DGRAM soit UDP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }
    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    // On prépare l'adresse d'attachement locale
    longueurAdresse = sizeof(struct sockaddr_in);
    memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
    pointDeRencontreLocal.sin_family = PF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // n'importe quelle interface
        locale disponible
    pointDeRencontreLocal.sin_port = htons(0); // l'os choisira un numéro de port libre

    // On demande l'attachement local de la socket
    if((bind(descripteurSocket, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse))
        < 0)
    {
        perror("bind");
        exit(-2);
    }
    printf("Socket attachée avec succès !\n");

    //<-- Début de l'étape n°3
    // Obtient la longueur en octets de la structure sockaddr_in
    longueurAdresse = sizeof(pointDeRencontreDistant);
    // Initialise à 0 la structure sockaddr_in

```

```

memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
// Renseigne la structure sockaddr_in avec les informations du serveur distant
pointDeRencontreDistant.sin_family = PF_INET;
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
// On choisit l'adresse IPv4 du serveur
inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier selon ses
    besoins

// Initialise à 0 le message
memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));

// Envoie un message au serveur
sprintf(messageEnvoi, "Hello world !\n");
ecrits = sendto(descripteurSocket, messageEnvoi, strlen(messageEnvoi), 0, (struct
    sockaddr *)&pointDeRencontreDistant, longueurAdresse);
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("sendto");
        close(descripteurSocket);
        exit(-3);
    case 0 :
        fprintf(stderr, "Aucune donnée n'a été envoyée !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* envoi de n octets */
        if(ecrits != strlen(messageEnvoi))
            fprintf(stderr, "Erreur dans l'envoi des données !\n\n");
        else
            printf("Message %s envoyé avec succès (%d octets)\n\n", messageEnvoi, ecrits);
}
//--> Fin de l'étape n°3 !

// On ferme la ressource avant de quitter
close(descripteurSocket);

return 0;
}

```

Étape n°3 : communication avec le serveur

Si vous testez ce client (sans serveur), vous risquez d'obtenir :

```

$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
    envoyé avec succès (14 octets)

```

Le message a été envoyé au serveur : ceci peut s'expliquer tout simplement parce que nous sommes en mode non-connecté.



Le protocole UDP ne prend pas en charge un mode fiable : ici, le client a envoyé des données sans savoir si un serveur était prêt à les recevoir !

Étape n°4 : vérification du bon fonctionnement de l'échange

Pour tester notre client, il nous faut quand même un serveur ! Pour cela, on va utiliser l'outil réseau `netcat` en mode serveur (-l) UDP (-u) sur le port 5000 (-p 5000).

```
$ nc -u -l -p 5000
```

Puis :

```
$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
  envoyé avec succès (14 octets)
```



Dans l'architecture client/serveur, on rappelle que c'est le client qui a l'initiative de l'échange. Il faut donc que le serveur soit en attente avant que le client envoie ses données.

Le message a bien été reçu et affiché par le serveur `netcat` :

```
$ nc -u -l -p 5000
Hello world !
```

Étape n°5 : réalisation d'un serveur UDP

Le code source d'un serveur UDP basique est très similaire à celui d'un client UDP. Évidemment, un serveur UDP a lui aussi besoin de créer une socket `SOCK_DGRAM` dans le domaine `PF_INET`. Puis, il doit utiliser l'appel système `bind()` pour lier sa socket d'écoute à une interface et à un numéro de port local à sa machine car le processus client doit connaître et fournir au moment de l'échange ces informations.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



Il est ici possible de préciser avec `INADDR_ANY` que toutes les interfaces locales du serveur accepteront les échanges des clients.

Dans notre exemple, le serveur va seulement réceptionner un datagramme en provenance du client. Pour cela, il va utiliser l'appel système `recvfrom()` :

```
$ man 2 recvfrom
```

RECV(2)

Manuel du programmeur Linux

RECV(2)

NOM

`recvfrom` - Recevoir un message sur une socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

DESCRIPTION

L'appel système `recvfrom()` est utilisé pour recevoir des messages.

Si `from` n'est pas NULL, et si le protocole sous-jacent fournit l'adresse de la source, celle-ci y est insérée. L'argument `fromlen` est un paramètre résultat, initialisé à la taille du tampon `from`, et modifié en retour pour indiquer la taille réelle de l'adresse enregistrée.

...

VALEUR RENVOYÉE

Ces fonctions renvoient le nombre d'octets reçus si elles réussissent, ou -1 si elles échouent. La valeur de retour sera 0 si le pair a effectué un arrêt normal.

Extrait de la page man de l'appel système `recvfrom`



C'est l'appel `recvfrom()` qui remplit la structure `sockaddr_in` avec les informations du point de communication du client (adresse IPv4 et numéro de port pour `PF_INET`).

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons, htonl et inet_aton */

#define PORT IPPORT_USERRESERVED // = 5000

#define LG_MESSAGE 256

int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreLocal;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
    int lus; /* nb d'octets lus */
    int retour;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0);

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
```

```
perror("socket"); // Affiche le message d'erreur
exit(-1); // On sort en indiquant un code erreur
}
printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

// On prépare l'adresse d'attachement locale
longueurAdresse = sizeof(struct sockaddr_in);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // n'importe quelle interface
    locale disponible
pointDeRencontreLocal.sin_port = htons(PORT); // <- 5000

// On demande l'attachement local de la socket
if((bind(descripteurSocket, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse))
    < 0)
{
    perror("bind");
    exit(-2);
}
printf("Socket attachée avec succès !\n");

//<-- Début de l'étape n°4
// Obtient la longueur en octets de la structure sockaddr_in
longueurAdresse = sizeof(pointDeRencontreDistant);
// Initialise à 0 la structure sockaddr_in (c'est l'appel recvfrom qui remplira cette
    structure)
memset(&pointDeRencontreDistant, 0x00, longueurAdresse);

// Initialise à 0 le message
memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));

// Réceptionne un message du client
lus = recvfrom(descripteurSocket, messageRecu, sizeof(messageRecu), 0, (struct sockaddr
    *)&pointDeRencontreDistant, &longueurAdresse);
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("recvfrom");
        close(descripteurSocket);
        exit(-3);
    case 0 :
        fprintf(stderr, "Aucune donnée n'a été reçue !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* réception de n octets */
        printf("Message %s reçu avec succès (%d octets)\n\n", messageRecu, lus);
}
//--> Fin de l'étape n°4 !

// On ferme la ressource avant de quitter
close(descripteurSocket);
```

```
    return 0;
}
```

Étape n°5 : réalisation d'un serveur UDP

Une simple exécution du serveur le place en attente d'une réception de données :

```
$ ./serveurUDP
Socket créée avec succès ! (3)
Socket attachée avec succès !
^C
```

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant ! Exécutons deux fois le même serveur et on obtient alors :

```
$ ./serveurUDP & ./serveurUDP
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket créée avec succès ! (3)
bind: Address already in use
```



Explication : l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur. TCP et UDP ne partagent pas le même espace d'adressage (numéro de port logique indépendant).

Testons notre serveur avec notre client :

```
$ ./serveurUDP
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
  reçu avec succès (14 octets)

$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
  envoyé avec succès (14 octets)
```



Il est évidemment possible de tester notre serveur avec le client UDP de netcat avec l'option -u.

Programmation Socket TCP (Windows)

Objectifs

L'objectif de cette partie est la mise en oeuvre d'une communication client/serveur en utilisant une socket TCP sous Windows.



Winsock (*W*indows *S*OCKet) est une bibliothèque logicielle pour Windows dont le but est d'implémenter une interface de programmation inspirée des sockets BSD. Son développement date de 1991 (Microsoft n'a pas implémenté Winsock 1.0.). Il y a peu de différences avec les sockets BSD, mais Winsock fournit des fonctions additionnelles pour être conforme au modèle de programmation Windows, par exemple les fonctions `WSAGetLastError()`, `WSAStartup()` et `WSACleanup()`.

Vous aurez besoin d'outils de compilation et fabrication de programmes sous Windows. **MinGW** (*M*inimalist *G*NU for *W*indows) est une adaptation des logiciels de développement et de compilation du GNU (**GCC** : *G*NU *C*ompiler *C*ollection) à la plate-forme Windows (Win32). Vous pouvez l'installer indépendamment ou intégrer à un EDI comme Dev-Cpp ou Code::Blocks.

Étape n°0 : préparation

Sous Windows, il faut tout d'abord initialiser avec `WSAStartup()` l'utilisation de la DLL Winsock par le processus. De la même manière, il faudra terminer son utilisation proprement avec `WSACleanup()`.

```
#include <winsock2.h>

// pour Visual Studio sinon ajouter -lws2_32
#pragma comment(lib, "ws2_32.lib")

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    /* ... */

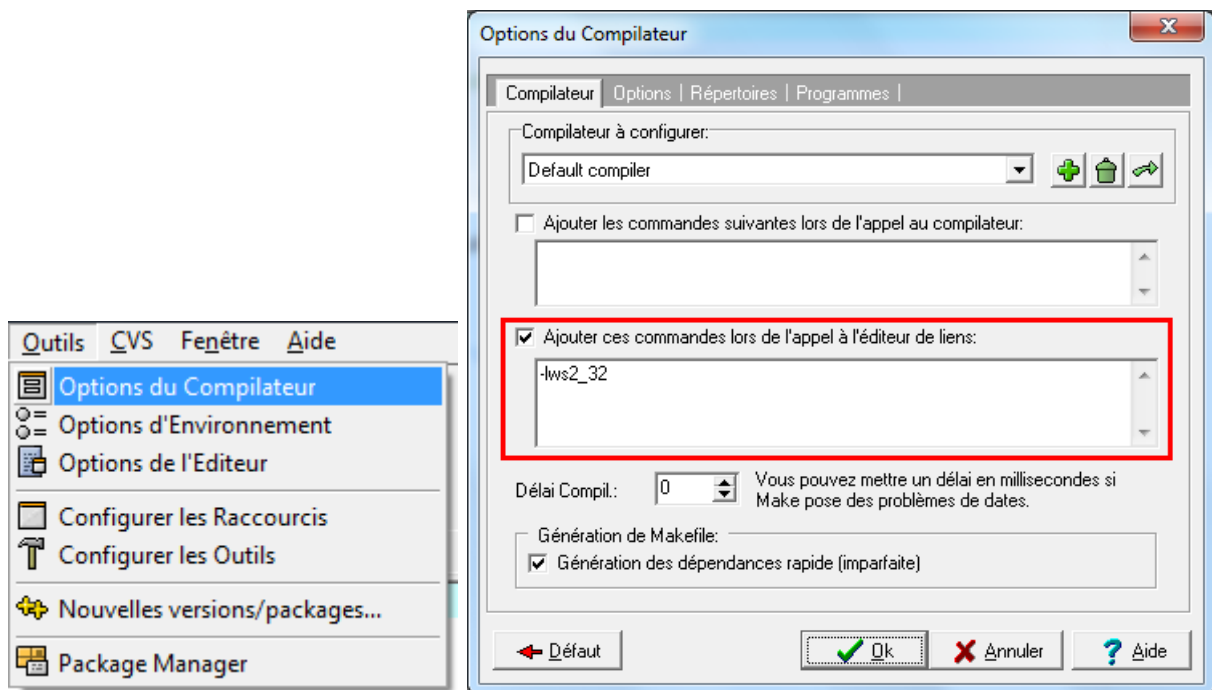
    WSACleanup(); // termine l'utilisation

    return 0;
}
```

Étape n°0 : préparation



Dans cet exemple, on utilise la version 2 de Winsock (`winsock2.h` et `ws2_32.lib`). Vous pouvez aussi utiliser la version 1 (`winsock.h` et `wsock32.lib`). Avec les compilateurs type GCC, il faudra ajouter `-lws2_32` à l'édition des liens.

Exemple : Lien vers la bibliothèque `ws2_32.lib` dans Dev-Cpp

Étape n°1 : création de la socket (côté client)

Pour créer une socket, on utilisera l'appel `socket()`. On commence par consulter la documentation associée à cet appel : [http://msdn.microsoft.com/en-us/library/ms740506\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms740506(v=vs.85).aspx).

The socket function creates a socket that is bound to a specific transport service provider.

```
SOCKET WINAPI socket(
    _In_ int af,
    _In_ int type,
    _In_ int protocol
);
```

...

Extrait de la documentation de l'appel socket

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    WSADATA WSAData; // variable initialisée par WSStartup

    WSStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    //-- Début de l'étape n°1 :
    SOCKET descripteurSocket;
    int iResult;

    // Crée un socket de communication
```

```

descripteurSocket = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
    protocole par défaut associé à SOCK_STREAM soit TCP */

if (descripteurSocket == INVALID_SOCKET)
{
    printf("Erreur creation socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

//--Fin de l'étape n°1 !
printf("Socket créée avec succès !\n");

// On ferme la ressource avant de quitter

iResult = closesocket(descripteurSocket);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation

return 0;
}

```

Étape n°1 : création de la socket (côté client)



Pour le paramètre `protocol`, on a utilisé la valeur 0 (voir commentaire). On aurait pu préciser le protocole TCP de la manière suivante : `IPPROTO_TCP`.

Étape n°2 : connexion au serveur

Maintenant que nous avons créé une socket TCP, il faut la connecter au processus serveur distant.

Pour cela, on va utiliser l'appel système `connect()`. On commence par consulter la documentation associée à cet appel : [http://msdn.microsoft.com/en-us/library/ms737625\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms737625(v=vs.85).aspx).

The connect function establishes a connection to a specified socket.

```

int connect(
    _In_ SOCKET s,
    _In_ const struct sockaddr *name,
    _In_ int namelen
);

...

```

Extrait de la page de documentation de l'appel connect

On rappelle que l'adressage du processus distant dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `AF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

L'interface socket propose une structure d'adresse générique :

```
struct sockaddr {
    ushort  sa_family;
    char    sa_data[14];
};

typedef struct sockaddr SOCKADDR;
```

La structure générique `sockaddr`

Et le domaine AF_INET utilise une structure compatible :

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)

struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct in_addr sin_addr;
    char     sin_zero[8];
};

typedef struct sockaddr_in SOCKADDR_IN;
```

La structure compatible `sockaddr_in` pour AF_INET

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations distantes du serveur (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `inet_addr()` pour convertir une **adresse IP** depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau)
- `htons()` pour convertir le **numéro de port** (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



L'ordre des octets du réseau est en fait *big-endian*. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 5000

int main()
{
    WSADATA WSAData; // variable initialisée par WSStartup

    WSStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0
```



```
SOCKET descripteurSocket;
int iResult;

// Crée un socket de communication
descripteurSocket = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
    protocole par défaut associé à SOCK_STREAM soit TCP */

if (descripteurSocket == INVALID_SOCKET)
{
    printf("Erreur creation socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

//--- Début de l'étape n°2 :
struct sockaddr_in pointDeRencontreDistant; // ou SOCKADDR_IN pointDeRencontreDistant;

// Renseigne la structure sockaddr_in avec les informations du serveur distant
pointDeRencontreDistant.sin_family = AF_INET;
// On choisit l'adresse IPv4 du serveur
pointDeRencontreDistant.sin_addr.s_addr = inet_addr("192.168.52.2"); // à modifier selon
    ses besoins
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreDistant.sin_port = htons(PORT); // = 5000

// Débute la connexion vers le processus serveur distant
iResult = connect(descripteurSocket, (SOCKADDR *)&pointDeRencontreDistant, sizeof(
    pointDeRencontreDistant));
if (iResult == SOCKET_ERROR)
{
    printf("Erreur connexion socket : %d\n", WSAGetLastError());
    iResult = closesocket(descripteurSocket); // On ferme la ressource avant de quitter
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    }
    WSACleanup();
    return 1; // On sort en indiquant un code erreur
}

//--- Fin de l'étape n°2 !
printf("Connexion au serveur réussie avec succès !\n");

// On ferme la ressource avant de quitter
iResult = closesocket(descripteurSocket);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
```

```
WSACleanup(); // termine l'utilisation  
  
return 0;  
}
```

Étape n°2 : connexion au serveur

Si vous testez ce client, vous risquez d'obtenir :

Erreur connexion socket : 10061

Ceci peut s'expliquer tout simplement parce qu'il n'y a pas de processus serveur à cette adresse !



La fonction `WSAGetLastError()` retourne seulement un code d'erreur. L'ensemble des codes d'erreurs sont déclarés dans le fichier d'en-tête `winsock2.h`. Par exemple ici, le code d'erreur 10061 correspond à l'étiquette `WSAECONNREFUSED` (connexion refusée). Pour obtenir le message associé à un code d'erreur, il faut utiliser la fonction `FormatMessageW()`.

```
wchar_t *s = NULL;  
FormatMessageW(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |  
    FORMAT_MESSAGE_IGNORE_INSERTS, NULL, WSAGetLastError(), MAKELANGID(LANG_NEUTRAL,  
    SUBLANG_DEFAULT), (LPWSTR)&s, 0, NULL);  
fprintf(stderr, "%S\n", s);  
LocalFree(s);
```

Affichage du message associé à un code d'erreur

Étape n°3 : vérification du bon fonctionnement de la connexion

Pour tester notre client, il nous faut un serveur ! Pour cela, on va utiliser l'outil réseau `netcat` en mode serveur (-l) sur le port 5000 (-p 5000) :

```
nc -l -p 5000
```

Puis en exécutant `clientTCPWin-2.exe`, on obtient :

Connexion au serveur réussie avec succès !



Dans l'architecture client/serveur, on rappelle que c'est le client qui a l'initiative de l'échange. Il faut donc que le serveur soit en écoute avant que le client fasse sa demande.

Étape n°4 : échange des données

On rappelle qu'une communication TCP est bidirectionnelle *full duplex* et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.



Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide ! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

Les fonctions d'échanges de données sur une socket TCP sont :

- `recv()` et `send()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket avec un paramètre `flags`



Les appels `recv()` et `send()` sont spécifiques aux sockets en mode connecté (TCP).

Faire communiquer deux processus sans aucun protocole de couche APPLICATION est tout de même difficile ! On va simplement fixer les règles d'échange suivantes :

- le client envoie en premier une chaîne de caractères
- et le serveur lui répondra "ok"

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 5000
#define LG_MESSAGE 256

int main()
{
    WSADATA WSAData; // variable initialisée par WSStartup

    WSStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    SOCKET descripteurSocket;
    int iResult;

    // Crée un socket de communication
    descripteurSocket = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (descripteurSocket == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    struct sockaddr_in pointDeRencontreDistant; // ou SOCKADDR_IN pointDeRencontreDistant;

    // Renseigne la structure sockaddr_in avec les informations du serveur distant
    pointDeRencontreDistant.sin_family = AF_INET;
    // On choisit l'adresse IPv4 du serveur
    pointDeRencontreDistant.sin_addr.s_addr = inet_addr("192.168.52.2"); // à modifier selon
        ses besoins
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreDistant.sin_port = htons(PORT); // = 5000

    // Débute la connexion vers le processus serveur distant
    iResult = connect(descripteurSocket, (SOCKADDR *)&pointDeRencontreDistant, sizeof(
        pointDeRencontreDistant));
```

```
if (iResult == SOCKET_ERROR)
{
    printf("Erreur connexion socket : %d\n", WSAGetLastError());
    iResult = closesocket(descripteurSocket); // On ferme la ressource avant de quitter
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    }
    WSACleanup();
    return 1; // On sort en indiquant un code erreur
}

printf("Connexion au serveur réussie avec succès !\n");

//--- Début de l'étape n°4 :
char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
int ecrits, lus; /* nb d'octets ecrits et lus */

sprintf(messageEnvoi, "Hello world !\n");
ecrits = send(descripteurSocket, messageEnvoi, (int)strlen(messageEnvoi), 0); // message
    à TAILLE variable
if (ecrits == SOCKET_ERROR)
{
    printf("Erreur envoi socket : %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Message %s envoyé avec succès (%d octets)\n\n", messageEnvoi, ecrits);

/* Reception des données du serveur */
lus = recv(ConnectSocket, messageRecu, sizeof(messageRecu), 0); /* attend un message de
    TAILLE fixe */
if( lus > 0 ) /* réception de n octets */
    printf("Message reçu du serveur : %s (%d octets)\n\n", messageRecu, lus);
else if ( lus == 0 ) /* la socket est fermée par le serveur */
    printf("La socket a été fermée par le serveur !\n");
else /* une erreur ! */
    printf("Erreur lecture socket : %d\n", WSAGetLastError());
//--- Fin de l'étape n°4 !

// On ferme la ressource avant de quitter
iResult = closesocket(descripteurSocket);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation
```

```
    return 0;
}
```

Étape n°4 : échange des données

On utilise la même procédure de test que précédemment en démarrant un serveur **netcat** sur le port 5000 :

```
nc -l -p 5000
```

Puis, on exécute notre client **clientTCPWin-3.exe** :

Connexion au serveur réussie avec succès !

Message Hello world !

envoyé avec succès (14 octets)

Message reçu du serveur : ok
(3 octets)

Dans la console où on a exécuté le serveur **netcat**, cela donne :

```
Hello world !
ok
```



Dans **netcat**, pour envoyer des données au client, il suffit de saisir son message et de valider par la touche Entrée.

Que se passe-t-il si le serveur s'arrête (en tapant **Ctrl-C** par exemple!) au lieu d'envoyer "ok" ?

```
Hello world !
^C
```

Dans la console du client **clientTCPWin-3.exe** :

Connexion au serveur réussie avec succès !

Message Hello world !

envoyé avec succès (14 octets)

La socket a été fermée par le serveur !

Notre client a bien détecté la fermeture de la socket côté serveur.

Dans les codes sources ci-dessus, nous avons utilisés l'appel **close()** pour fermer la socket et donc la communication. En TCP, la communication étant bidirectionnelle *full duplex*, il est possible de fermer plus finement l'échange en utilisant l'appel **shutdown()** :

The shutdown function disables sends or receives on a socket.

```
int shutdown(
    _In_ SOCKET s,
    _In_ int how
);
```

s [in] : A descriptor identifying a socket.

`how [in]` : A flag that describes what types of operation will no longer be allowed. Possible values for this flag are listed in the `Winsock2.h` header file.

`SD_RECEIVE` : Shutdown receive operations.
0

`SD_SEND` : Shutdown send operations.
1

`SD_BOTH` : Shutdown both send and receive operations.
2

...

Extrait de la documentation de l'appel shutdown

Étape n°5 : réalisation d'un serveur TCP

Évidemment, un serveur TCP a lui aussi besoin de créer une socket `SOCK_STREAM` dans le domaine `AF_INET`. Mis à part cela, le code source d'un serveur TCP basique est très différent d'un client TCP dans le principe. On va détailler ces différences étape par étape.

On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de processus client. Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.

Pour mettre en oeuvre cela, le serveur va utiliser l'appel `bind()` qui va lui permettre de lier sa socket d'écoute à une interface et à un numéro de port local à sa machine.

The `bind` function associates a local address with a socket.

```
int bind(  
    _In_ SOCKET s,  
    _In_ const struct sockaddr *name,  
    _In_ int namelen  
);
```

...

Extrait de la documentation de l'appel bind

On rappelle que l'adressage d'un processus (local ou distant) dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `AF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

Rappel : l'interface socket propose une structure d'adresse générique `sockaddr` et le domaine `AF_INET` utilise une structure compatible `sockaddr_in`.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



Normalement il faudrait indiquer l'adresse IPv4 de l'interface locale du serveur qui acceptera les demandes de connexions. Il est ici possible de préciser avec `INADDR_ANY` que toutes les interfaces locales du serveur accepteront les demandes de connexion des clients.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000

int main()
{
    WSADATA WSADATA; // variable initialisée par WSStartup

    WSStartup(MAKEWORD(2,0), &WSADATA); // indique la version utilisée, ici 2.0

    SOCKET socketEcoule;
    int iResult;

    //--- Début de l'étape n°5 :
    // Crée un socket de communication
    socketEcoule = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (socketEcoule == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // On prépare l'adresse d'attachement locale
    struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;

    // Renseigne la structure sockaddr_in avec les informations locales du serveur
    pointDeRencontreLocal.sin_family = AF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
        locales disponibles
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

    iResult = bind(socketEcoule, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
        pointDeRencontreLocal));
    if (iResult == SOCKET_ERROR)
    {
```

```

    printf("Erreur bind socket : %d\n", WSAGetLastError());
    closesocket(socketEcoule);
    WSACleanup();
    return 1;
}

//--- Fin de l'étape n°5 !
printf("Socket attachée avec succès !\n");

// On s'endort ... (cf. test)
sleep(2);

// On ferme la ressource avant de quitter
iResult = closesocket(socketEcoule);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation

return 0;
}

```

Étape n°5 : réalisation d'un serveur TCP

Le test est concluant :

Socket attachée avec succès !

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant !
Exécutons deux fois le même serveur et on obtient alors :

bind: Address already in use



Explication : l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur.

Étape n°6 : mise en attente des connexions

Maintenant que le serveur a créé et attaché une socket d'écoute, il doit la placer en attente passive, c'est-à-dire capable d'accepter les demandes de connexion des processus clients.

Pour cela, on va utiliser l'appel `listen()` :

The `listen` function places a socket in a state in which it is listening for an incoming connection.

```

int listen(
    _In_ SOCKET s,
    _In_ int backlog
);

```


s [in] : A descriptor identifying a bound, unconnected socket.

backlog [in] : The maximum length of the queue of pending connections. If set to SOMAXCONN, the underlying service provider responsible for socket s will set the backlog to a maximum reasonable value. There is no standard provision to obtain the actual backlog value.

...

Extrait de la documentation de l'appel listen



Si la file est pleine, le serveur sera dans une situation de DOS (*Deny Of Service*) car il ne peut plus traiter les nouvelles demandes de connexion.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000

int main()
{
    WSADATA WSAData; // variable initialisée par WSStartup

    WSStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    SOCKET socketEcoule;
    int iResult;

    // Crée un socket de communication
    socketEcoule = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (socketEcoule == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // On prépare l'adresse d'attachement locale
    struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;

    // Renseigne la structure sockaddr_in avec les informations locales du serveur
    pointDeRencontreLocal.sin_family = AF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
        locales disponibles
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

    iResult = bind(socketEcoule, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
        pointDeRencontreLocal));
```

```
if (iResult == SOCKET_ERROR)
{
    printf("Erreur bind socket : %d\n", WSAGetLastError());
    closesocket(socketEcoule);
    WSACleanup();
    return 1;
}

printf("Socket attachée avec succès !\n");

//--- Début de l'étape n°6 :
// On fixe la taille de la file d'attente (pour les demandes de connexion non encore
    traitées)
if (listen(socketEcoule, SOMAXCONN) == SOCKET_ERROR)
{
    printf("Erreur listen socket : %d\n", WSAGetLastError());
}

//--- Fin de l'étape n°6 !
printf("Socket placée en écoute passive ...\n");

// On ferme la ressource avant de quitter
iResult = closesocket(socketEcoule);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation

return 0;
}
```

Étape n°6 : mise en attente des connexions

Étape n°7 : accepter les demandes connexions

Cette étape est cruciale pour le serveur. Il lui faut maintenant accepter les demandes de connexion en provenance des processus client.

Pour cela, il va utiliser l'appel `accept()` :

The `accept` function permits an incoming connection attempt on a socket.

```
SOCKET accept(
    _In_     SOCKET s,
    _Out_    struct sockaddr *addr,
    _Inout_  int *addrlen
);
```

`s [in]` : A descriptor that identifies a socket that has been placed in a listening state with the `listen` function. The connection is actually made with the socket that is

returned by accept.

addr [out] : An optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the addr parameter is determined by the address family that was established when the socket from the sockaddr structure was created.

addrlen [in, out] : An optional pointer to an integer that contains the length of structure pointed to by the addr parameter.

...

Extrait de la documentation de l'appel accept



Explication : imaginons qu'un client se connecte à notre socket d'écoute. L'appel `accept()` va retourner une nouvelle socket connectée au client qui servira de socket de dialogue. La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.

Le principe est simple mais un problème apparaît pour le serveur : comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ? Il y a plusieurs solutions à ce problème notamment la programmation multi-tâche car ici le serveur a besoin de paralléliser plusieurs traitements.

On va pour l'instant ignorer ce problème et mettre en oeuvre un serveur basique : c'est-à-dire mono-client (ou plus exactement un client après l'autre) !

Concernant le dialogue, on utilisera les mêmes fonctions `recv()/send()` que le client.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000
#define LG_MESSAGE 256

int main()
{
    WSADATA WSAData; // variable initialisée par WSASStartup

    WSASStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    SOCKET socketEcoule;
    int iResult;

    // Crée un socket de communication
    socketEcoule = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (socketEcoule == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
}
```

```

}

// On prépare l'adresse d'attachement locale
struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;

// Renseigne la structure sockaddr_in avec les informations locales du serveur
pointDeRencontreLocal.sin_family = AF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
    locales disponibles
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

iResult = bind(socketEcoute, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
    pointDeRencontreLocal));
if (iResult == SOCKET_ERROR)
{
    printf("Erreur bind socket : %d\n", WSAGetLastError());
    closesocket(socketEcoute);
    WSACleanup();
    return 1;
}

printf("Socket attachée avec succès !\n");

// On fixe la taille de la file d'attente (pour les demandes de connexion non encore
    traitées)
if (listen(socketEcoute, SOMAXCONN) == SOCKET_ERROR)
{
    printf("Erreur listen socket : %d\n", WSAGetLastError());
}

printf("Socket placée en écoute passive ...\n");

//--- Début de l'étape n°7 :
SOCKET socketDialogue;
struct sockaddr_in pointDeRencontreDistant;
int longueurAdresse = sizeof(pointDeRencontreDistant);
char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
int ecrits, lus; /* nb d'octets ecrits et lus */

// boucle d'attente de connexion : en théorie, un serveur attend indéfiniment !
while(1)
{
    memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
    memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));
    printf("Attente d'une demande de connexion (quitter avec Ctrl-C)\n\n");
    // c'est un appel bloquant
    socketDialogue = accept(socketEcoute, (SOCKADDR *)&pointDeRencontreDistant, &
        longueurAdresse);

    if (socketDialogue == INVALID_SOCKET)
    {

```

```

    printf("Erreur accept socket : %d\n", WSAGetLastError());
    closesocket(socketEcoule);
    WSACleanup();
    return 1;
}

// On réception les données du client (cf. protocole !)
// ici appel bloquant
lus = recv(socketDialogue, messageRecu, sizeof(messageRecu), 0); /* attend un
    message de TAILLE fixe */
if( lus > 0 ) /* réception de n octets */
    printf("Message reçu du client : %s (%d octets)\n\n", messageRecu, lus);
else if ( lus == 0 ) /* la socket est fermée par le serveur */
    printf("socket fermé\n");
else /* une erreur ! */
    printf("Erreur lecture socket : %d\n", WSAGetLastError());

// On envoie des données vers le client (cf. protocole !)
sprintf(messageEnvoi, "ok\n");
ecrits = send(socketDialogue, messageEnvoi, (int)strlen(messageEnvoi), 0); //
    message à TAILLE variable
if (ecrits == SOCKET_ERROR)
{
    printf("Erreur envoi socket : %d\n", WSAGetLastError());
    closesocket(socketDialogue);
    WSACleanup();
    return 1;
}

printf("Message %s envoyé (%d octets)\n\n", messageEnvoi, ecrits);

// On ferme la socket de dialogue et on se replace en attente ...
closesocket(socketDialogue);
}

//--- Fin de l'étape n°7 !
// On ferme la ressource avant de quitter
iResult = closesocket(socketEcoule);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation

return 0;
}

```

Étape n°7 : accepter les demandes connexions

Testons notre serveur `serveurTCPWin-3.exe` avec notre client :

Socket attachée avec succès !

Socket placée en écoute passive ...

Attente d'une demande de connexion (quitter avec Ctrl-C)

Message reçu : Hello world !
(14 octets)

Message ok
envoyé (3 octets)

Attente d'une demande de connexion (quitter avec Ctrl-C)

Message reçu : Hello world !
(14 octets)

Message ok
envoyé (3 octets)

Attente d'une demande de connexion (quitter avec Ctrl-C)

^C

On va exécuter deux clients à la suite :

Connexion au serveur réussie avec succès !
Message Hello world !
envoyé avec succès (14 octets)

Message reçu du serveur : ok
(3 octets)

Connexion au serveur réussie avec succès !
Message Hello world !
envoyé avec succès (14 octets)

Message reçu du serveur : ok
(3 octets)



Il est évidemment possible de tester notre serveur avec des clients TCP existants comme telnet ou netcat.

Programmation TCP (Linux) IPv6

L'API *socket* reste identique mais on utilisera `PF_INET6` pour l'appel `socket()`.

Évidemment il y a une structure d'adresse adaptée pour IPv6 :

```
// Dans /usr/include/linux/in6.h

struct in6_addr {
    union {
        __u8    u6_addr8[16];
        __be16   u6_addr16[8];
        __be32   u6_addr32[4];
    } in6_u;
#define s6_addr    in6_u.u6_addr8
#define s6_addr16  in6_u.u6_addr16
#define s6_addr32  in6_u.u6_addr32
};

struct sockaddr_in6 {
    unsigned short int sin6_family; /* AF_INET6 */
    __be16            sin6_port;    /* Transport layer port # */
    __be32            sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr    sin6_addr;   /* IPv6 address */
    __u32             sin6_scope_id; /* scope id (new in RFC2553) */
};
```

Remarque : la nouvelle structure d'adresse `sockaddr_in6` a une longueur de 28 octets. Elle est donc plus grande que le type générique `sockaddr`. Pour corriger cela, une nouvelle structure `sockaddr_storage` a été définie, de taille suffisante afin de pouvoir prendre en compte tous les protocoles supportés.

Il y a aussi des nouvelles fonctions de conversion :

- `getaddrinfo()` (équivalent à `gethostbyname()`, `gethostbyaddr()`, `getservbyname()`, `getservbyport()`)
- `getnameinfo()` (équivalent à `gethostbyaddr()` et `getservbyport()`)
- `inet_pton()` et `inet_ntop()` (équivalent à `inet_addr()` et `inet_ntoa()`)

Remarques :

- on appellera `getaddrinfo()` avant la création de la socket avec `socket()` ;
- L'adresse *wildcard* : `IN6ADDR_ANY_INIT` (équivalent à `INADDR_ANY`) ;
- L'adresse de *loopback* : `IN6ADDR_LOOPBACK_INIT` (équivalent à `INADDR_LOOPBACK`).

Exemple de **client IPv6** :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>

#define LG_MESSAGE 256
```

```
int main(int argc, char **argv)
{
    int sock;
    int ret;
    struct addrinfo *res;
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */

    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s host port\n", argv[0]);
        exit(1);
    }

    struct addrinfo hints = {
                                0,
                                PF_INET6,
                                SOCK_STREAM,
                                0,
                                0,
                                NULL,
                                NULL,
                                NULL
                            };

    // Traduction d'adresse
    ret = getaddrinfo(argv[1], argv[2], &hints, &res);
    if (ret)
    {
        fprintf(stderr, "Erreur getaddrinfo: %s\n", gai_strerror(ret));
        exit(1);
    }

    // Création de la socket
    if ((sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0)
    {
        freeaddrinfo(res);
        perror("socket");
        exit(1);
    }

    // Connexion
    if (connect(sock, res->ai_addr, res->ai_addrlen) < 0)
    {
        close(sock);
        freeaddrinfo(res);
        perror("connect");
        exit(1);
    }
    freeaddrinfo(res);

    // Lecture de données
    int lus = read(sock, messageRecu, LG_MESSAGE*sizeof(char));
```



```
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("read");
        close(sock);
        exit(1);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le serveur !\n\n");
        close(sock);
    default: /* réception de n octets */
        printf("Message reçu du serveur : %s (%d octets)\n\n", messageRecu, lus);
        close(sock);
}

exit(0);
}
```

Client TCP IPv6

Test du **client IPv6** avec un serveur netcat :

```
$ nc -6 -l -p 5000
hello
```

```
$ ./a.out ip6-localhost 5000
Message reçu du serveur : hello
(6 octets)
```

Exemple de **serveur IPv6** :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>

#define LG_MESSAGE 256

int main(int argc, char **argv)
{
    int sock; // socket d'écoute
    int ret;
    struct addrinfo *res;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */

    if (argc < 1)
    {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }
}
```

```
struct addrinfo hints = {
    AI_PASSIVE, // utilisera IN6ADDR_ANY_INIT pour bind
    PF_INET6,
    SOCK_STREAM,
    0,
    0,
    NULL,
    NULL,
    NULL
};

// ou :
/*
memset(&hints, 0, sizeof hints);
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
hints.ai_family = PF_INET6;
*/

// Traduction d'adresse
ret = getaddrinfo(NULL, argv[1], &hints, &res);
if (ret)
{
    fprintf(stderr, "Erreur getaddrinfo: %s\n", gai_strerror(ret));
    exit(1);
}

// Création de la socket
if ((sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0)
{
    freeaddrinfo(res);
    perror("socket");
    exit(1);
}

// Attachement local de la socket
if (bind(sock, res->ai_addr, res->ai_addrlen) < 0)
{
    perror("bind");
    close(sock);
    exit(1);
}

// Place la socket en écoute passive
// et fixe la taille de la file d'attente des demandes de connexion non encore traitées
listen(sock, 5); // ou SOMAXCONN (égal à 128)

freeaddrinfo(res);

int s; // socket de dialogue
struct sockaddr_storage from; // adresse distante (client)
int len = sizeof(from);
```

```
// Attente de connexion
s = accept(sock, (struct sockaddr *)&from, &len);
if (s < 0)
{
    perror("accept");
    close(sock);
    close(s);
    exit(1);
}

char host[NI_MAXHOST], service[NI_MAXSERV];
// Lecture de la requête
int lus = read(s, messageRecu, LG_MESSAGE*sizeof(char));
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("read");
        close(s);
        close(sock);
        exit(1);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "socket fermée par le client !\n");
        close(s);
        close(sock);
        exit(0);
    default: /* réception de n octets */
        ret = getnameinfo((struct sockaddr *)&from, len, host, NI_MAXHOST, service,
            NI_MAXSERV, NI_NUMERICSERV);
        if (ret == 0)
        {
            printf("Reçu %d octets de %s:%s\n", lus, host, service);
            printf("Requête reçue du client : %s\n", messageRecu);
        }
        else
            fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));
}

// Réponse du serveur
sprintf(messageEnvoi, "ok\n");
int ecrits = write(s, messageEnvoi, strlen(messageEnvoi));
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("write");
        close(s);
        close(sock);
        exit(1);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "socket fermée par le client !\n");
        close(s);
        close(sock);
        exit(0);
    default: /* envoi de n octets */
```

```
        break;
    }

    close(s);
    close(sock);
    exit(0);
}
```

Serveur TCP IPv6

Test du **serveur IPv6** avec un client **netcat** :

```
$ ./a.out 5000
Reçu 6 octets de ip6-localhost:58300
Requête reçue du client : hello

$ nc -6 ip6-localhost 5000
hello
ok
```

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

Question 1. Qu'est-ce qu'une *socket* ?

Question 2. Quelles sont les trois caractéristiques d'une *socket* ?

Question 3. Quelles sont les deux informations qui définissent un point de communication en IPv4 ?

Question 4. Comment le serveur connaît-il le port utilisé par le client ?

Question 5. Comment le client connaît-il le port utilisé par le serveur ?

Question 6. À quelle couche du modèle DoD est reliée l'interface de programmation *socket* ?

Question 7. Quel protocole de niveau Transport permet d'établir une communication en mode connecté ?

Question 8. Quel protocole de niveau Transport permet d'établir une communication en mode non-connecté ?

Question 9. Quel est l'ordre des octets en réseau ?

Question 10. À quels protocoles correspond le domaine `PF_INET` ? Est-ce le seul utilisable avec l'interface *socket* ? En citer au moins un autre.

Annexe 1 : le résolveur nom → adresse

Parfois, on ne connaît pas l'adresse IPv4 d'un serveur mais seulement son nom (par exemple `localhost` ou `www.google.fr`). Il faut alors un **résolveur** pour obtenir son adresse IPv4.

Avant on utilisait la fonction `gethostbyname()` mais, celle-ci est maintenant obsolète et, les programmes doivent utiliser `getaddrinfo()` à la place.

Exemple pour une socket TCP :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit et atoi */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <netdb.h> /* pour getaddrinfo */

int main(int argc, char *argv[])
{
    int descripteurSocket;
    int retour;
    struct addrinfo hints;
    struct addrinfo *result, *rp;

    // Il faut 2 arguments à ce programme
    if (argc != 3)
    {
        fprintf(stderr, "Erreur : argument manquant !\n");
        fprintf(stderr, "Usage : %s adresse_ip numero_port\n", argv[0]);
        exit(-1); // On sort en indiquant un code erreur
    }

    // Obtenir un point de communication distant correspondant au couple nom/port fourni en
    // argument
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = PF_INET; // sinon AF_UNSPEC pour IPv4 ou IPv6
    hints.ai_socktype = SOCK_STREAM; // mode connecté
    hints.ai_flags = 0; // options supplémentaires, ici aucune
    hints.ai_protocol = 0; // n'importe quel type

    retour = getaddrinfo(argv[1], argv[2], &hints, &result);
    if (retour != 0)
    {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(retour)); // Affiche le message d'
        // erreur
        exit(-2); // On sort en indiquant un code erreur
    }

    // getaddrinfo() retourne une liste de structures d'adresses
    // on cherche la première qui permettra un connect()
    for (rp = result; rp != NULL; rp = rp->ai_next)
    {
        // Crée un socket de communication
```

```
descripteurSocket = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);

// Échec ?
if (descripteurSocket == -1)
    continue; // alors on essaye la suivante

// Débute la connexion vers le processus serveur distant
// Ok ?
if((connect(descripteurSocket, rp->ai_addr, rp->ai_addrlen)) != -1)
    break; // on a trouvé une adresse valide !

// On ferme cette ressource avant de passer à la suivante
close(descripteurSocket);
}

if (rp == NULL)
{
    // on a donc trouvé aucune adresse valide !
    fprintf(stderr, "Impossible de se connecter à cette adresse !\n");
    exit(-3); // On sort en indiquant un code erreur
}

freeaddrinfo(result); // on libère les résultats obtenus et on continue ...

printf("Socket créée avec succès ! (%d)\n", descripteurSocket);
printf("Connexion au serveur réussie avec succès !\n");

// ...

// On ferme la ressource avant de quitter
close(descripteurSocket);

return 0;
}
```